Vérification de systèmes paramétrés avec Cubicle

Sylvain Conchon[1,2] & Alain Mebsout[1,2] & Fatiha Zaïdi[1]

1: Université Paris Sud, CNRS, F-91405 Orsay
2: INRIA Saclay – Île-de-France F-91893 Orsay
conchon@lri.fr, mebsout@lri.fr, zaidi@lri.fr

Résumé

Cubicle est un model-checker pour vérifier des propriétés de sûreté d'algorithmes faisant intervenir un nombre quelconque de processus. Ces algorithmes sont décrits sous forme de systèmes de transitions dits *paramétrés*. Les propriétés de sûreté ainsi que les états du système sont décrits par des formules logiques du premier ordre. Pour vérifier ces propriétés, Cubicle met en œuvre un algorithme d'atteignabilité par chaînage arrière qui utilise un démonstrateur SMT (Satisfiabilité Modulo Théories). Les expériences, menées sur la vérification d'algorithmes d'exclusion mutuelle et de protocoles de cohérence de cache, montrent que Cubicle est efficace et compétitif avec les *model checkers* de la même famille. Cubicle est un logiciel libre développé en OCaml. Il utilise le démonstrateur Alt-Ergo ZERO, une version allégée et améliorée d'Alt-Ergo. Une implantation parallèle se reposant sur la bibliothèque Functory est également proposée.

1. Introduction

Le *model checking* consiste à s'assurer qu'un modèle, représentant un programme ou l'abstraction d'un système complexe, vérifie certaines propriétés. Les outils de vérification par *model checking* sont nombreux et diffèrent selon la nature des modèles à analyser, les techniques pour les représenter et enfin les propriétés à vérifier. Ainsi, on distingue par exemple les *model checkers* qui analysent des programmes dans un langage comme C (Blast [21]) ou Java (JavaPathFinder [20]) de ceux qui manipulent des automates (Spin [22]) ou des systèmes de transition avec un nombre d'états fini (Murφ [14]) ou infini (TReX [4]). La représentation des états peut être énumérative (les états sont alors représentés de manière individuelle) ou symbolique (dans ce cas, on manipule des ensembles d'états, par exemple à l'aide de BDD comme dans NuSMV [10]). On distingue également le type des propriétés à vérifier (sûreté, vivacité ou équité) ainsi que les logiques pour les exprimer (LTL, CTL, CTL* etc.).

Cubicle [13] appartient à la famille des *model checkers* symboliques qui vérifient des propriétés de sûreté sur des systèmes de transition infinis et faisant intervenir un nombre quelconque de processus. Ces systèmes sont dits *paramétrés*. Par exemple Cubicle est capable de prouver des propriétés d'exclusion mutuelle d'un algorithme à n threads pour tout n. Un des avantages de cette approche paramétrée est qu'il est souvent bien plus facile de prouver une propriété pour un nombre arbitraire de processus que de prouver cette propriété pour un nombre fixé lorsque celui-ci devient grand. Malheureusement, vérifier la sûreté d'un système paramétré est en général indécidable [5]. Aussi, il est important d'identifier un fragment sur lequel il est possible de prouver les propriétés qui nous intéressent. La classe des systèmes traitée par Cubicle est celle des *systèmes à tableaux* [16] dont la sûreté est décidable sous certaines conditions [17]. C'est une classe syntaxiquement restreinte de systèmes de transition paramétrés dont les états sont représentés par un ensemble fini de tableaux indexés par un nombre arbitraire d'identificateurs de processus. Les protocoles de cohérence de cache et les algorithmes d'exclusion mutuelle sont des exemples typiques de systèmes dont les états peuvent

être représentés avec des tableaux. Cubicle vérifie la sûreté de tels programmes de la manière suivante : les états initiaux ainsi que les états *dangereux* du système sont représentés par des formules logiques et l'analyse consiste à vérifier qu'aucun état *dangereux* n'est atteignable à partir d'un état initial en appliquant les instances des transitions du système. Pour cela, cet algorithme calcule une clôture de la pré-image de la formule *dangereuse* par la relation de transition à l'aide d'un démonstrateur SMT. Ce calcul est rendu possible car on manipule des formules spécifiques (conjonctions de littéraux existentiellement quantifiées) représentant un ensemble infini d'états qu'on appelle *cubes*. La sûreté est garantie si aucun de ces cubes ne contient un des états initiaux.

L'algorithme d'atteignabilité de Cubicle est directement issu du cadre théorique proposé par Ghilardi et Ranise [17]. Cependant, il faut aller au delà de ces travaux si on souhaite prouver la sûreté de systèmes paramétrés donnés comme défis à la communauté du model checking. Par exemple, la vérification du protocole de cohérence de cache proposé par Steve German, et décrit dans [7], nécessite l'implantation de nombreuses optimisations. Dans cet article, on présente les détails et les optimisations importantes qui permettent à Cubicle de prouver la sûreté d'un tel protocole.

La famille des model checkers pour systèmes paramétrés n'est pas très fournie mais elle compte parmi ses rangs des outils qui travaillent sur différents fragments décidables, tels que MCMT [18], Undip [3] ou PFS [2]. Le plus proche concurrent de Cubicle est MCMT (fondé sur les mêmes travaux théoriques). Tout en étant aussi compétitif, Cubicle propose une implantation libre, un langage d'entrée simple d'usage ainsi qu'une architecture parallèle.

Cet article étend la présentation de Cubicle faite dans [13] en donnant plus de détails sur l'implémentation et les propriétés sur lesquelles se basent nos optimisations (en particulier, les critères pour garantir la terminaison de l'algorithme d'atteignabilité). Nous présentons également l'interface OCaml d'Alt-Ergo ZERO, le démonstrateur automatique SMT utilisé dans Cubicle.

Cet article présente le langage d'entrée de Cubicle en section 2 et les détails d'implantation en section 3. Son architecture parallèle, décrite en section 4, repose sur la bibliothèque Functory [15]. Cubicle utilise le démonstrateur SMT Alt-Ergo ZERO, une version allégée et améliorée d'Alt-Ergo [11] qui se présente sous la forme d'une bibliothèque OCaml autonome décrite en section 5. Cubicle est écrit en OCaml et est disponible sous licence Apache à l'adresse `http://cubicle.lri.fr`.

2. Langage d'entrée

Le langage d'entrée de Cubicle est une version typée du langage de Murφ [14] et similaire à UCLID [8]. Bien que limité pour l'instant, il est assez expressif pour permettre de décrire aisément des systèmes paramétrés conséquents (75 transitions, 40 variables et tableaux pour le protocole FLASH [23] par exemple).

La description d'un système dans Cubicle commence par des déclarations de types, de variables globales et de tableaux. Cubicle connaît quatre types en interne : le type des entiers (`int`), le type des réels (`reals`), le type des booléens (`bool`) et le type des identificateurs de processus (`proc`). Les tableaux ont la contrainte qu'ils doivent être indexés par des variables de type `proc`. L'utilisateur a aussi la liberté de déclarer ses propres types abstraits ou ses propres types énumérés. L'exemple suivant définit un type énuméré `state` à trois constructeurs `Idle`, `Want` et `Crit` ainsi qu'un type abstrait `data`.

```
type state = Idle | Want | Crit
type data
```

Dans ce qui suit on déclare une variable globale `Timer` de type `real` et trois tableaux indexés par le type `proc`.

```
var Timer : real
array State[proc] : state
array Chan[proc] : data
array Flag[proc] : bool
```

Les états initiaux du système sont définis par une conjonction de littéraux, universellement quantifiée. Ces littéraux caractérisent les valeurs de certains tableaux et variables. Dans un souci de simplicité, l'exemple suivant définit les états initiaux comme ceux ayant leur tableau `Flag` à `False` pour tout processus `z`, `State` à `Idle` et leur variable globale `Timer` valant `0.0`.

```
init(z) { Flag[z] = Flase && State[z] = Idle && Timer = 0.0 }
```

Les propriétés de sûreté à vérifier sont exprimées sous forme négative comme des formules caractérisant les états *dangereux*. Chaque formule dangereuse (ou *unsafe*) doit être un *cube*, i.e. être de la forme $\exists \bar{x}.(distinct(\bar{x}) \wedge C)$, où $\bar{x}$ est un ensemble de variables représentant des identificateurs de processus, $distinct(\bar{x})$ est la conjonction des différences entre les variables de $\bar{x}$ (exprimant que ces variables doivent être deux à deux distinctes) et C est une conjonction de littéraux. Dans la syntaxe concrète, les quantificateurs existentiels sont laissés implicites, ainsi que le $distinct(\bar{x})$. La formule *dangereuse* suivante exprime que les mauvais états du système sont ceux où il existe deux processus distincts `x` et `y` tels que le tableau `State` contienne la valeur `Crit` à ces deux indices.

```
unsafe(x y) { State[x] = Crit && State[y] = Crit }
```

Le reste du système est donné comme un ensemble de transitions de la forme garde/action. Chaque transition peut être paramétrée par un ou plusieurs identificateurs de processus comme dans l'exemple suivant.

```
transition t (i j)
requires { i < j && State[i] = Idle && Flag[i] = Flase &&
           forall_other k. (Flag[k] = Flag[j] || State[k] <> Want) }
{
  Timer := Timer + 1.0;
  Flag[i] := True;
  State[k] := case
              | k = i : Want
              | State[k] = Crit && k < i : Idle
              | _ : State[k];
}
```

Ici, les paramètres `i` et `j` de la transition sont implicitement quantifiés existentiellement et doivent être les identificateurs de processus deux à deux distincts. Les gardes sont des conjonctions de littéraux (équations, différences, inéquations) et de formules universellement quantifiées de la forme $\forall k.C_1 \vee \ldots \vee C_n$ où k est une variable processus universellement quantifiée distincte de tous les paramètres de la transition et $C_1, \ldots, C_n$ sont des conjonctions de littéraux. Chaque action est une mise à jour d'une variable globale ou d'un tableau. La sémantique des transitions veut que ces

mises à jour soient réalisées de manière atomique et donc chaque variable qui apparaît à droite d'un signe `:=` dénote la valeur de cette variable avant la transition. Les mises à jour de tableaux sont codées soit comme de simples affectations `Flag[i] := True` soit par des constructions par cas comme `State[k] := case ...` où la variable `k` est implicitement universellement quantifiée. Dans cette construction `case`, chaque condition doit être une conjonction de littéraux et suppose la négation de toutes les conditions précédentes. Le cas par défaut est dénoté par `_` .

Cette relation de transition (décrite par l'ensemble des transitions) définit l'exécution du système comme une boucle infinie qui à chaque itération :

1. choisit de manière non déterministe une instance de transition dont la garde est vraie dans l'état courant du système
2. met à jour les variables et tableaux d'états conformément aux actions de la transition choisie

Un système est *sûr* si aucun des états dangereux ne peut être atteint à partir d'un des états initiaux.

3. Implantation d'une boucle d'atteignabilité symbolique efficace

Cubicle implante une boucle d'atteignabilité par chaînage arrière pour vérifier les propriétés de sûreté des systèmes à tableaux. Étant donné un programme défini par une formule initiale $\mathcal{I}$ (décrivant les états initiaux du programme), un ensemble de transitions $\mathcal{T}$ et une formule $\mathcal{U}$ représentant les états dangereux, l'algorithme est donnée par le pseudo-code suivant :

```
1  init : V ← ∅
2         Q ← ∅
3         push_queue(Q,U)
4  while not_empty(Q) do
5      φ ← pop_queue(Q)
6      if ¬(φ ∧ I ⊢ ⊥) then return(Unsafe)   (* sûreté *)
7      if ¬(φ ⊢ ⋁_{ψ∈V} ψ) then              (* point fixe local*)
8          V ← V ∪ φ
9          push_queue(Q, pre_T(φ))
10 done
11 return(Safe)
```

FIGURE 1 – Algorithme d'atteignabilité arrière

L'algorithme en figure 1 maintient un ensemble V des nœuds *visités* et une file de priorité Q des nœuds *non visités*. Initialement, V est vide et Q contient la formule dangereuse du système. Ensuite, à chaque itération de la boucle, le cube φ ayant la plus grande priorité est sorti de la file Q et sa sûreté est vérifiée en testant sa cohérence avec la formule initiale (ligne 6). Si ce test de sûreté réussit, alors on enchaîne un test de *subsomption* (ou point fixe local) $\neg(\varphi \wedge \mathcal{I} \vdash \bot)$ (ligne 7). Si ce dernier test échoue, le cube φ est ajouté à l'ensemble des nœuds visités V et on calcule la formule $\texttt{pre}_{\mathcal{T}}(\varphi)$ (ligne 9). Cette formule représente l'ensemble des états à partir desquels il est possible d'atteindre un des états représentés par φ en une transition, autrement dit la pré-image de φ par la relation de transition $\mathcal{T}$. Une des propriété des systèmes à tableaux (indexés par des variables processus) est que si φ est un cube alors $\texttt{pre}_{\mathcal{T}}(\varphi)$ est une disjonction (union) de cubes. Ces cubes sont ensuite ajoutés à la file Q et on répète une nouvelle itération de la boucle. Au contraire, si le test de *subsomption* est concluant alors φ est ignoré car cela signifie que chaque état représenté par φ est aussi représenté par au moins un cube qu'on a déjà visité. L'algorithme termine soit lorsque un test de sûreté échoue ou bien lorsque la file Q devient vide.

Les tests de satisfiabilité sont envoyés à un démonstrateur SMT (Satisfiabilité Modulo Théories). Les tests de sûreté (ligne 6) sont faciles à prouver car il s'agit simplement de formules closes obtenues par skolémisation. Par contre, les tests de subsomption (ligne 7) de la forme $\neg(\varphi \vdash \bigvee_{\psi \in V} \psi)$ sont plus difficiles. En effet, prouver la validité de l'implication $\varphi \vdash \bigvee_{\psi \in V} \psi$ revient à montrer que la formule $\neg(\neg\varphi \vee \bigvee_{\psi \in V} \psi)$, c'est-à-dire $\varphi \wedge \bigwedge_{\psi \in V} \neg\psi$, est insatisfiable. Puisque φ et ψ sont des formules existentiellement quantifiées de la forme $\exists \bar{x}.F$ et $\exists \bar{y}.G_\psi$ (où F et G_ψ représentent des conjonctions de littéraux), cela revient à manipuler une formule contenant une conjonction de clauses universellement quantifiées $F \wedge \bigwedge_{\psi \in V} \forall \bar{y}.\neg G_\psi$. Le cadre théorique des systèmes à tableaux nous garantit qu'il suffit de considérer l'ensemble Σ des substitutions de $\bar{y}$ vers $\bar{x}$. Par exemple, si on suppose que V contient 20 000 nœuds et que $|\bar{x}| = |\bar{y}| = 5$, il faut alors construire une formule $H = F \wedge \bigwedge_{\psi \in V} \bigwedge_{\sigma \in \Sigma} \forall \bar{y}.\neg G_\psi \sigma$ avec 2,4 millions de clauses. Malgré leur efficacité il n'est pas envisageable d'envoyer de telles formules à un démonstrateur SMT.

Pour passer ce test de subsomption, Cubicle essaye de montrer que H est insatisfiable en la construisant et en la vérifiant incrémentalement. Ceci est fait en examinant toutes les paires (ψ, σ) une à une puis en ajoutant les $\neg G_\psi \sigma$ à la formule H jusqu'à ce qu'elle devienne insatisfiable. Pour chaque paire (ψ, σ), on vérifie si le cube $G_\psi \sigma$ a une des propriétés suivantes :

Proposition 3.1.

1. *si $G_\psi \sigma$ contient un littéral qui contredit directement un littéral de F alors $G_\psi \sigma$ est redondant dans H pour le test qui nous intéresse et peut donc être enlevé de H*
2. *si $G_\psi \sigma$ est un sous ensemble de F alors H est insatisfiable*

Démonstration.

1. Soit $G_\psi \sigma = g \wedge G'$ et $F = f \wedge F'$ tels que $g \wedge f \vdash \bot$. Autrement dit $f \implies \neg g$, et donc $(f \wedge F') \wedge (\neg g \vee \neg G')$ se réduit en $(f \wedge F')$.
2. Soit $G_\psi \sigma$ un sous ensemble de F, alors $F = G_\psi \sigma \wedge F'$ donc $F \wedge \neg G_\psi \sigma$ est trivialement insatisfiable, donc H est insatisfiable.

□

Avec la première propriété, on vérifie que le cube $G_\psi \sigma$ n'est pas redondant avant même d'appliquer la substitution σ ; s'il y a redondance le cube est ignoré et une nouvelle paire (ψ, σ) est traitée. Si le cube n'est pas redondant, on lui fait passer le test d'inclusion $G_\psi \sigma \subset F$ de la seconde propriété. Pour calculer efficacement ces tests ensemblistes, les cubes sont représentés à l'aide d'une simple structure de tableau OCaml. Si l'inclusion est vérifiée alors H est déclarée comme étant insatisfiable ; sinon on ajoute $\neg G_\psi \sigma$ à H et le démonstrateur SMT vérifie si la nouvelle version (renforcée) de H devient insatisfiable.

L'intégration de Cubicle avec le démonstrateur SMT au niveau de l'interface de programmation est cruciale pour traiter efficacement ces tests de subsomption. En pratique, on utilise un seul contexte du démonstrateur pour chacun de ces tests ; il est seulement enrichi incrémentalement avec les $\neg G_\psi \sigma$ et sa cohérence est vérifiée à chaque itération. Pour garantir une mise en œuvre la plus efficace et la plus complète possible des tests d'inclusion, les cubes sont maintenus sous forme normale, i.e. les variables sont renommées et les simplifications possibles sont réalisées lors de la construction.

La stratégie d'exploration de l'espace de recherche est elle aussi essentielle. Cubicle gagnera bien souvent à explorer le moins de nœuds possible, ce qui suggère de donner la priorité aux cubes les plus *généraux*, i.e. ceux qui représentent les ensembles d'états les plus grands. Après ce constat, il est clair que des stratégies de recherche naïves comme l'exploration en largeur ou l'exploration en profondeur ne sont pas adaptées. Par défaut, Cubicle utilise une forme de recherche en largeur (BFS) combinée avec une heuristique de retardement pour certains cubes. Dans la stratégie par défaut, un cube est retardé s'il introduit de nouvelles variables de processus ou s'il n'apporte pas d'information supplémentaire sur au moins un tableau. Ces stratégies peuvent être changées en passant différents arguments sur la

ligne de commande : l'option `-search` permet de choisir une autre stratégie de recherche comme la recherche en profondeur (DFS) ou une de ses variantes et l'option `-postpone` permet de changer ou de désactiver l'heuristique de retardement. Enfin Cubicle est capable de supprimer des parties de l'arbre de recherche déjà visité, donc des cubes de V, lorsque ceux-ci deviennent subsumés par un nœud plus récent (figure 2).

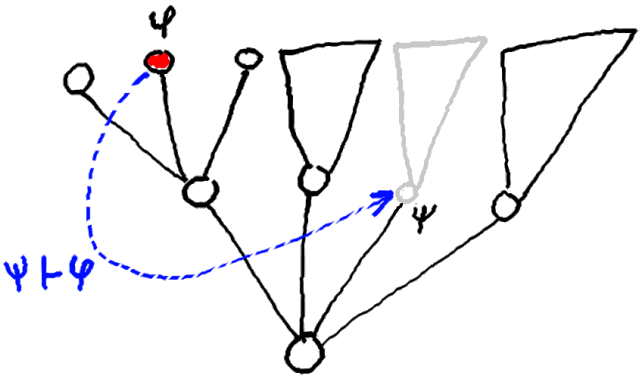

FIGURE 2 – Suppression, *a posteriori*, du nœud ψ (et de ses fils) subsumé par φ

Afin d'aider Cubicle à converger plus rapidement, il est possible d'ajouter des invariants (exprimées à l'aide de *cubes*) qui vont permettre de réduire l'exploration de grandes parties de l'arbre de recherche. Ces propriétés peuvent être ajoutées de deux manières différentes :

- soit comme *candidats invariants*, et elles seront alors *vérifiées* par Cubicle ;
- soit comme *invariants*, et elles seront alors simplement *supposées* par Cubicle. Dans ce cas, la correction du model checker peut être remise en cause si l'invariant fourni est faux.

Cubicle dispose également d'un mécanisme pour synthétiser automatiquement des *candidats* invariants. Chacun de ces candidats est vérifié en exécutant une instance de la boucle de model checking dont les ressources machine sont limitées volontairement (en limitant le nombre de nœuds visités par exemple). En plus de ces invariants synthétisés de manière dynamique, Cubicle découvre des *invariants de sous-typage* à l'aide d'une analyse statique qui calcule, pour chaque variable dont le type est un type énuméré, un sous ensemble des valeurs possibles pour cette variable. Ces invariants sont ensuite envoyés directement au démonstrateur SMT qui supporte la définition de sous-types pour les types énumérés.

Terminaison. La terminaison de l'algorithme d'atteignabilité de Cubicle n'est pas garantie en général, mais on peut montrer qu'elle peut être obtenue sous certaines conditions.

Pour étudier la terminaison de l'algorithme en figure 1, on s'intéresse à l'évolution de l'ensemble V des nœuds visités. Plus précisément, on considère la séquence d'inclusions $V_0 \subseteq V_1 \subseteq \cdots \subseteq V_n \subseteq \cdots$ où V_n représente l'ensemble V à la $n^{\text{ième}}$ itération de la boucle `while`. Pour que la boucle d'atteignabilité ne termine pas, il faut nécessairement que de nouveaux cubes soient ajoutés régulièrement dans V. Par conséquent, il existe une infinité d'ensembles de nœuds visités V_{k_i} tels que $V_{k_1} \subset V_{k_2} \subset \cdots$. On va définir les condiditions suffisantes pour que cette sous-séquence d'inclusions *strictes* soit finie. Le lecteur est renvoyé à [1, 17] pour plus de détails sur les propositions et théorèmes suivants.

Étant donné un système paramétré à tableaux $\mathcal{S}$, on appelle *configuration* de $\mathcal{S}$ un état concret du système, c'est-à-dire un modèle pour les types, les variables globales et les tableaux du système. En particulier, une configuration doit fixer le nombre de processus du système (*i.e.* la cardinalité du type `proc`) qui peut être utilisé pour indéxer les tableaux. Un système à tableaux $\mathcal{S}$ a un nombre potentiellement infini de configurations.

Étant donné un cube φ manipulé par Cubicle, on note $[\![\varphi]\!]$ l'ensemble des configurations qui satisfont φ. Si l'ensemble des configurations est muni d'un pré-ordre bien fondé $\preceq$ (*i.e.* une relation binaire réflexive et transitive sans suite infinie strictement décroissante), alors on peut montrer que $[\![\varphi]\!]$ est un *idéal*, c'est-à-dire que si $s \in [\![\varphi]\!]$ et $s \preceq s'$ alors $s' \in [\![\varphi]\!]$. Par extension, il est immédiat de montrer que chaque ensemble V_n de nœuds visités est également un idéal.

Maintenant, si $\preceq$ a également la propriété d'être un *bel ordre*, c'est-à-dire si pour toute séquence infinie de configurations $s_1, s_2, \ldots$, il existe nécessairement $i < j$ tels que $s_i \preceq s_j$, alors la terminaison de l'algorithme est assurée par le théorème suivant :

Théorème 3.2. *Si $\preceq$ est un bel ordre, alors toute séquence d'inclusions strictes d'idéaux $[\![V_{k_1}]\!] \subset [\![V_{k_2}]\!] \subset \cdots$ est finie.*

De plus, $[\![.]\!]$ est monotone, *i.e.* si $V \subset V'$ alors $[\![V]\!] \subset [\![V']\!]$ donc la finitude de la séquence d'inclusion $[\![V_{k_1}]\!] \subset [\![V_{k_2}]\!] \subset \cdots$ implique bien la finitude de la séquence d'inclusion $V_0 \subseteq V_1 \subseteq \cdots \subseteq V_n \subseteq \cdots$ calculée par l'algorithme. La preuve du théorème 3.2 repose sur l'argument suivant : par contradiction, si cette séquence est infinie, alors il existe également une séquence infinie de configurations $s_1, s_2, \ldots$ telle que $s_i \in [\![V_{k_i}]\!]$ et $s_i \notin [\![V_{k_j}]\!]$, pour tout $j < i$. De plus, $s_j \not\preceq s_i$, sinon $s_i \in [\![V_{k_j}]\!]$ puisque chaque $[\![V_{k_j}]\!]$ est un idéal. L'existence de cette suite infinie contredit donc l'hypothèse que $\preceq$ est un bel ordre.

Il ne reste donc plus qu'à déterminer les conditions nécessaires pour que l'ensemble des configurations puisse être muni d'un *bel ordre*. Cela dépend essentiellement du choix des opérations de comparaison sur le type `proc` et sur les types des éléments des tableaux.

- Si les éléments des tableaux appartiennent à un type énuméré et que les indices sont seulement munis de la relation d'égalité, alors il est possible d'exhiber un bel-ordre sur les configurations par le lemme de Dickson [17, 26].
- Si les éléments des tableaux appartiennent à un type énuméré et que les indices sont munis de la relation d'égalité et d'un ordre total, alors il est possible d'exhiber un bel-ordre sur les configurations par le lemme de Higman [17, 26].
- Si les éléments des tableaux sont des rationnels et que les indices sont seulement munis de la relation d'égalité, alors il est possible d'exhiber un bel-ordre sur les configurations par le lemme de Kruskal [17, 26].

4. Architecture parallèle

Une manière naturelle d'augmenter la rapidité des model checkers est de paralléliser les tâches qui demandent du temps de calcul pour tirer parti de la disponibilité grandissante des machines multi-cœurs ou multi-processeurs ainsi que des clusters de machines [19, 6, 24]. Dans le cadre de Cubicle cette parallélisation peut être faite de façon immédiate au niveau de la génération d'invariants car la boucle de model checking qui vérifie ces derniers est complètement indépendante du reste de la recherche. De manière plus intéressante, la boucle d'atteignabilité arrière peut elle même être parallélisée à un certain degré. Une implantation directe d'une boucle parallèle affecterait cependant l'orientation de l'exploration, et casserait les heuristiques employées et pourrait même rendre certaines optimisations de la section 3 non sûres. De plus nos expériences ont montré qu'une exploration non déterministe de l'espace de recherche est souvent moins efficace qu'une recherche guidée.

Dans notre cas, les tâches qui consomment le plus de ressources sont les tests de subsomption (ligne 7 de l'algorithme figure 1) qui peuvent être des problèmes difficiles même pour des démonstrateurs SMT modernes, principalement de par leur taille. Pour paralléliser Cubicle nous avons implanté une version concurrente de la recherche en largeur (BFS) en se reposant sur la simple observation que tous les calculs à faire sur toutes les branches à un même niveau de l'arbre de recherche peuvent être effectués en parallèle. Cette implantation utilise une architecture centralisée de type maître/esclave.

Le maître attribue des tests de point fixe aux esclaves et une barrière de synchronisation est placée à la fin de chaque niveau de l'arbre pour conserver une exploration en largeur. Le maître calcule ensuite de manière asynchrone les pré-images des nœuds qui n'ont pas pu être vérifiés comme étant des points fixes par les esclaves. Pendant ce temps le maître peut aussi attribuer des tâches de génération d'invariants qui seront traitées si des esclaves deviennent disponibles. Maintenant, pour supprimer des nœuds subsumés a posteriori de V de manière à ne pas se retrouver dans le cas de la figure 3, le maître doit simplement ignorer les résultats concernant les nœuds qui ont été supprimés pendant qu'un esclave vérifiait leur propriété de point fixe.

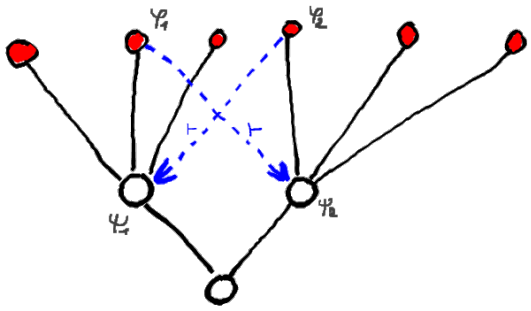

FIGURE 3 – Une mauvaise synchronisation des tests de subsomption effectués en parallèle (flèches en bleu) peut supprimer des nœuds de manière incorrecte (ici, une des deux suppressions par subsomption doit être ignorée pour garantir la sûreté)

D'un point de vue technique, Cubicle fournit une exploration parallèle de l'espace de recherche en utilisant n processus concurrents sur une architecture multi-cœurs lorsqu'il est invoqué avec l'option `-j` n. L'implantation utilise Functory [15], une bibliothèque OCaml fournissant une interface fonctionnelle riche et polymorphe et permet facilement de distribuer des calculs parallèles. Functory permet d'utiliser des architectures multi-cœurs ainsi que des réseaux de machines et fournit un mécanisme robuste de tolérance aux pannes. Bien que Cubicle ne fournisse pas aujourd'hui d'implantation distribuée, c'est une des directions envisagées pour l'évolution du logiciel qui demanderait tout de même d'être capable de limiter la taille des données devant circuler sur le réseau lors des communications entre le maître et les esclaves. Ici c'est la taille des nœuds visités V qui peut rapidement devenir un goulot d'étranglement dans une architecture distribuée reposant sur l'échange de messages. Une solution à ce problème est que chaque esclave maintienne sa propre copie de V et que seules les mises à jour de cet ensemble soient transmises.

5. La bibliothèque Alt-Ergo ZERO

Cubicle se sert de manière intensive d'un démonstrateur SMT pour décharger ses tests de sûreté et de subsomption. Pour cela, il est distribué avec son propre démonstrateur qu'il invoque, automatiquement, au travers d'une API OCaml. Ce démonstrateur, dérivé d'Alt-Ergo et baptisé Alt-Ergo ZERO, est aussi distribué sous la forme d'une bibliothèque OCaml disponible librement sur le web à l'adresse `http://cubicle.lri.fr/alt-ergo-zero/`. La documentation complète au format Ocamldoc est disponible sur le web à la même adresse. Nous décrivons ici brièvement l'utilisation de cette bibliothèque.

Alt-Ergo ZERO prend en entrée des formules logiques sans quantificateurs combinant des opérateurs de la logique propositionnelle avec des opérateurs prédéfinis pour les théories de l'égalité, de l'arithmétique linéaire (entiers et rationnels) et des types énumérés.

Pour des raisons d'efficacité, Alt-Ergo ZERO utilise des chaînes de caractères hash-consées [12] pour un partage maximal (et une opération de comparaison efficace), dont l'utilisation est aussi documentée mais ne sera pas décrite dans cette section. L'API du démonstrateur se divise en six modules qui fournissent des fonctionnalités de typage (`Type`, `Symbol` et `Variant`), de construction de termes (`Term`) et de formules (`Formula`), et d'appel au démonstrateur (`Solver`).

5.1. Typage

Les types de base des entiers, des réels, des booléens et des processus sont fournis directement, mais il est toujours possible de déclarer un nouveau type abstrait en appelant la fonction `Type.declare "mon_type" []` ou de définir un type énuméré ayant comme constructeurs `A` et `B` en appelant `Type.declare "mon_type" ["A"; "B"]`. Le module `Type` permet aussi de récupérer les constructeurs d'un type.

```
module Type : sig
  type t = Hstring.t

  val type_int : t
  val type_real : t
  val type_bool : t
  val type_proc : t

  val declare : Hstring.t -> Hstring.t list -> unit
  val constructors : t -> Hstring.t list
  ...
end
```

Le module `Symbol` suivant permet quant à lui de déclarer des symboles de fonction qui, s'ils ne prennent aucun argument en entrée seront des constantes. Le deuxième argument de `Symbol.declare` est la liste des types de ses arguments et le troisième est son type de retour. Ce module permet aussi de faire différentes requêtes sur les types des symboles.

```
module Symbol : sig
  type t = Hstring.t

  val declare : Hstring.t -> Type.t list -> Type.t -> unit
  val type_of : t -> Type.t list * Type.t
  val has_abstract_type : t -> bool
  val has_type_proc : t -> bool
  val declared : t -> bool
end
```

Enfin le module `Variant` permet de faire une analyse de sous-typage. Pour utiliser cette fonctionnalité, il faut tout d'abord initialiser les types des symboles avec la fonction `Variant.init`, puis ajouter une à une les contraintes de sous typage que l'on désire voir respectées. Ensuite, un seul appel à la fonction `Variant.close` suffit à calculer les types les plus petits possibles et à mettre à jour l'information apportée par ces raffinements dans l'environnement de typage global.

```
module Variant : sig
  val init : (Symbol.t * Type.t) list -> unit
  val close : unit -> unit
  val assign_constr : Symbol.t -> Hstring.t -> unit
  val assign_var : Symbol.t -> Symbol.t -> unit
  ...
end
```

5.2. Construction de termes et formules

Pour construire des termes arithmétiques, Alt-Ergo ZERO fournit les opérateurs classiques ainsi que des fonctions pour créer des constantes numériques et des applications de fonction.

```
module Term : sig
  type t
  type operator = Plus | Minus | Mult | Div | Modulo

  val make_int : Num.num -> t
  val make_real : Num.num -> t
  val make_app : Symbol.t -> t list -> t
  val make_arith : operator -> t -> t -> t

  val is_int : t -> bool
  val is_real : t -> bool

  val t_true : t
  val t_false : t
end
```

Les formules sont soit des littéraux construits avec la fonction `make_lit` soit des combinaisons de littéraux construits avec la fonction `make`.

```
module Formula : sig
  type comparator = Eq  | Neq | Le  | Lt
  type combinator = And | Or  | Imp | Not
  type t =
    | Lit of Literal.LT.t
    | Comb of combinator * t list

  val f_true : t
  val f_false : t

  val make_lit : comparator -> Term.t list -> t
  val make : combinator -> t list -> t
  ...
end
```

```
exception Unsat of int list

module type Solver = sig
  type state
  ...
  val clear : unit -> unit
  val save_state : unit -> state
  val restore_state : state -> unit

  val assume : ?profiling:bool -> id:int -> Formula.t -> unit
  val check : ?profiling:bool -> unit -> unit
  val entails : ?profiling:bool -> id:int -> Formula.t -> bool
end

module Make (Dummy : sig end) : Solver
```

FIGURE 4 – Interface du démonstrateur d'Alt-Ergo ZERO

5.3. Utiliser le démonstateur SMT

L'interface du démonstrateur d'Alt-Ergo ZERO est donnée dans la figure 4. Elle se présente comme un foncteur `Make` qui prend en argument un module avec une signature vide. Cette interface fonctorisée est utilisée pour permettre la création de plusieurs instances de démonstrateur. Chaque démonstrateur obtenu par l'application de ce foncteur est impératif. L'état interne est représenté par un type abstrait et deux fonctions `save_state` et `restore_state` qui permettent respectivement de sauvegarder et de restaurer cet état à volonté. La fonction `assume` permet d'ajouter une formule au contexte et de lui donner un identifiant. La mise en forme normale conjonctive est faite à ce moment et les propagations unitaires possibles sont effectuées. Il est possible à tout moment de vérifier le contexte du démonstrateur en appelant la fonction `check` ce qui lancera réellement Alt-Ergo ZERO. Si le contexte du démonstrateur est incohérent alors l'exception `Unsat` est levée, accompagnée d'une liste d'entiers correspondant au *unsat core* (ou noyau d'insatisfiabilité) sous la forme d'identifiants associés aux formules supposées avec la fonction `assume`. Cette interface fournit aussi une fonction `entails` pour vérifier si le contexte du démonstrateur implique une formule donnée, sans en changer l'état interne.

L'exemple ci-dessous montre comment utiliser Alt-Ergo ZERO pour déterminer la (non) satisfiabilité de la conjonction de littéraux suivante :

$$f(x+3) = u \wedge f(y+2) = w \wedge x = y - 1 \wedge u \neq w$$

où `t` est un type abstrait, f un symbole de fonction de type `int` $\rightarrow$ `t`, x et y deux entiers et u et w deux variables de type `t`

On commence par créer une instance du démonstrateur (et quelques raccourcis pour gagner en visibilité) de la manière suivante :

```
open Smt
module S = Symbol
module T = Term
module F = Formula
module Solver = Make (struct end)
```

Puis, on déclare le type abstrait `t` à l'aide de la fonction `Type.declare`. Le nom du type est simplement défini à l'aide d'une chaîne *hash consées* `"t"` :

```
let type_t = Type.declare (Hstring.make "t") []
```

Ensuite, on déclare les symboles de la formule à l'aide de la fonction `Symbol.declare` :

```
let x = S.declare (Hstring.make "x") [] type_int
let y = S.declare (Hstring.make "y") [] type_int
let u = S.declare (Hstring.make "u") [] type_t
let w = S.declare (Hstring.make "w") [] type_t
let f = S.declare (Hstring.make "f") [Type.type_int] type_t
```

Enfin, chaque sous-terme de la formule est défini à l'aide des fonctions `Term.make_app`, `Term.make_int` et `Term.make_arith` :

```
let tx = T.make_app x []
let ty = T.make_app y []
let tu = T.make_app u []
let tw = T.make_app w []
let t3 = T.make_int (Num.Int 3) []
let t2 = T.make_int (Num.Int 2) []
let t1 = T.make_int (Num.Int 1) []
let fx3 = T.make_app f (T.make_arith T.Plus tx t3)
let fy2 = T.make_app f (T.make_arith T.Plus ty t2)
```

Chaque littéral est ensuite défini à l'aide de la fonction `Formula.make_lit` de la manière suivante :

```
let l1 = F.make_lit F.Eq [fx3; tu]                          (* f(x + 3) = u *)
let l2 = F.make_lit F.Eq [fy2; tw]                          (* f(y + 2) = w *)
let l3 = F.make_lit F.Eq [tx; (T.make_arith T.Minus ty t1)] (* x = y - 1 *)
let neg_goal = F.make_lit F.Neq [tu; tw]                    (* u <> w *)
```

Les littéraux sont enfin ajoutés les uns après les autres au contexte de l'instance du démonstrateur à l'aide de la fonction `Solver.assume`. Puis on vérifie la satisfiabilité du contexte final en appelant la fonction `Solver.check` :

```
try
  Solver.clear ();
  Solver.assume ~id:1 l1;
  Solver.assume ~id:2 l2;
  Solver.assume ~id:3 l3;
  Solver.assume ~id:4 neg_goal;
  Solver.check ();
  print_endline "satisfiable"
with Unsat _ -> print_endline "unsatisfiable"
```

6. Expériences

Nous avons évalué Cubicle sur des algorithmes d'exclusion mutuelle et des protocoles de cohérence de cache classiques mais aussi sur des protocoles plus difficiles à prouver (grand nombre de transitions, de variables, etc.). Dans le tableau figure 5, on compare les performances de Cubicle avec d'autres model checkers existants pour systèmes paramétrés. Toutes les expériences ont été réalisées sur une machine 64 bits avec un processeur quadri-cœurs Intel® Xeon® cadencé à 3,2 GHz et comportant 24 Go de

mémoire. Pour chaque outil, on donne les résultats obtenus avec les meilleurs réglages qu'on ait trouvé. Il est à noter que la version concurrente de Cubicle a été lancée sur quatre cœurs (i.e. avec l'option `-j 4`). Cette version n'a été exécutée que sur les exemples qui prennaient un temps significatif (> 10 secondes) en séquentiel. On a marqué par X les *benchmarks* qu'il nous a été impossible de traduire à cause de restrictions syntaxiques.

	Cubicle		MCMT [18]	Undip [3]	PFS [2]
	seq.	4 cœurs			
bakery	0.01s	–	0.01s	0.04s	0.01s
Dijkstra	0.24s	–	0.99s	**0.04s**	0.26s
Distributed_Lamport	**2.3s**	–	12.7s	unsafe	X
Java_Mlock	0.04s	–	0.06s	0.25s	0.02s
Ricart_Agrawala	**1.8s**	–	1m12s	4.3s	X
Szymanski_at	**0.12s**	–	0.71s	13.5s	timeout
Berkeley	0.01s	–	0.01s	0.01s	0.01s
flash_aggregated [25]	0.01s	–	0.02s	0.01s	X
German_Baukus	**25.0s**	17.1s	3h39m	9m43s	X
German_pfs	**6m23s**	3m8s	11m31s	timeout	47m22s
German_undip	**0.17s**	–	0.57s	1m32	X
Illinois	0.02s	–	0.04s	0.06s	0.06s
Moesi	0.01s	–	0.01s	0.01s	0.01s

FIGURE 5 – Benchmarks

Ces résultats sont très prometteurs. En effet, ils montrent tout d'abord que la version séquentielle de Cubicle est compétitive et que la version parallèle est capable d'atteindre une accélération d'environ $1,8$ sur quatre cœurs. C'est un bon résultat si on considère que toutes les unités de calcul ne peuvent pas être utilisées à leur maximum à cause des barrières de synchronisation requises pour garder une stratégie pertinente. En pratique, on a remarqué que les meilleures performances étaient obtenues en utilisant toutes les optimisations décrites dans la section 3 (à l'exception de la génération d'invariants qui peut demander beaucoup de temps pour des résultats incertains). Dans le tableau figure 6, on met en évidence les effets des différentes optimisations sur une version du protocole German extrait de [7]. La colonne "permut." désigne le calcul des permutations pertinentes et "suppression" désigne la suppression des nœuds subsumés *a posteriori*. En particulier il est intéressant de remarquer que l'analyse statique de sous-typage améliore les performances d'un ordre de grandeur sur cet exemple.

Optimisations				**Temps réel** (nb. de nœuds)	
permut.	suppression	sous-typage	génération d'invariants	séquentiel	4 cœurs
Non	Non	Non	Non	∞	∞
Oui	Non	Non	Non	50m8s (22580)	27m13s (20710)
Oui	**Oui**	Non	Non	35m16s (20405)	19m39s (19685)
Oui	**Oui**	Non	**Oui**	20m45s (15089)	13m55s (14527)
Oui	**Oui**	**Oui**	Non	**25.0s** (3322)	**17.1s** (3188)

FIGURE 6 – Effets des différentes optimisations sur le protocole German

Le code Cubicle complet du protocole German est donné dans l'annexe A.

7. Conclusion et perspectives

Nous avons présenté Cubicle, un model checker capable de prouver des propriétés de sûreté de systèmes de transition paramétrés. Son langage d'entrée permet notamment de décrire des algorithmes d'exclusion mutuelle et des protocoles de cohérence de cache intéressants comme le German ou le FLASH [23]. Les expériences menées montrent que Cubicle est très compétitif par rapport aux model checkers de la même famille. Cubicle utilise la bibliothèque Functory pour son architecture parallèle et la bibliothèque SMT Alt-Ergo ZERO, une version allégée et améliorée d'Alt-Ergo.

Les perspectives envisagées pour l'évolution de Cubicle concernent à la fois son expressivité et son efficacité. En terme d'expressivité, nous souhaitons étendre le langage d'entrée avec tout d'abord, un langage procédural de plus haut niveau (fonctions, séquences, boucles, primitives de communications, threads). Ensuite, nous envisageons une extension du langage logique pour décrire plus largement les formules définissant les états initiaux. Enfin la définition des types peut être étendue avec des structures de données comme les enregistrements ou les types sommes. Pour réaliser cela, le démonstrateur Alt-Ergo ZERO devra aussi intégrer de nouvelles procédures de décision.

Concernant l'efficacité de Cubicle, le prochain défi est la preuve de sûreté du protocole FLASH [23], considéré comme un des plus complexes protocoles de cohérence de cache académiques. Il comporte 600 millions d'états accessibles lorsque seulement quatre processus sont mis en jeu. Encore aujourd'hui, peu de méthodes sont capables de prouver la sûreté d'un tel protocole et toutes requièrent une intervention humaine [9, 27]. En comparaison, le protocole German compte 40 000 états pour quatre processus. Pour faire la preuve de sûreté de FLASH, et donc espérer pouvoir aborder des protocoles de taille industrielle, il est nécessaire de chercher des techniques pour réduire l'espace d'états de Cubicle, et améliorer son mécanisme de génération d'invariants. Ceci fait l'objet d'un travail en cours.

La bibliothèque OCaml Alt-Ergo ZERO a été conçue pour être autonome et utilisable dans d'autres contextes. Par exemple, on envisage de l'utiliser pour développer un autre type de model checker par k-induction et également de s'en servir dans des outils de test. Pour cela Alt-Ergo ZERO doit être étendu pour renvoyer des modèles servant à construire des traces et contre-exemples avec des variables instanciées.

Références

[1] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321, 1996.

[2] Parosh Aziz Abdulla, Giorgio Delzanno, Noomene Ben Henda, and Ahmed Rezine. Regular model checking without transducers (On efficient verification of parameterized systems). In *TACAS*, pages 721–736, 2007.

[3] Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV*, pages 145–157, 2007.

[4] Aurore Annichini, Ahmed Bouajjani, and Mihaela Sighireanu. TReX : A tool for reachability analysis of complex systems. In *CAV*, pages 368–372. 2001.

[5] Krzysztof Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22 :307–309, 1986.

[6] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE : Parallel Distributed Model Checker (Tool paper). In *HiBi/PDMC*, pages 4–7, 2010.

[7] Kai Baukus, Yassine Lakhnech, and Karsten Stahl. Parameterized verification of a cache coherence protocol : Safety and liveness. In *VMCAI*, pages 317–330, 2002.

[8] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *CAV*, pages 78–92, 2002.

[9] Ching-Tsun Chou, Phanindra Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In *FMCAD*, pages 382–398. 2004.

[10] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2 : An opensource tool for symbolic model checking. In *CAV*, pages 241–268. 2002.

[11] Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. CC(X) : Semantic combination of congruence closure with solvable theories. *ENTCS*, 198(2) :51–69, 2008.

[12] Sylvain Conchon and Jean-Christophe Filliâtre. Type-Safe Modular Hash-Consing. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, September 2006.

[13] Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Cubicle : A parallel SMT-based model checker for parameterized systems. In *CAV*, pages 718–724. 2012.

[14] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *ICCD*, pages 522–525, 1992.

[15] Jean-Christophe Filliâtre and K. Kalyanasundaram. Functory : A distributed computing library for Objective Caml. In *TFP*, pages 65–81, 2011.

[16] Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. Towards SMT model checking of array-based systems. In *IJCAR*, pages 67–82, 2008.

[17] Silvio Ghilardi and Silvio Ranise. Backward reachability of array-based systems by SMT solving : Termination and invariant synthesis. *LMCS*, 6(4), 2010.

[18] Silvio Ghilardi and Silvio Ranise. MCMT : A model checker modulo theories. In *IJCAR*, pages 22–29, 2010.

[19] Orna Grumberg, Tamir Heyman, Nili Ifergan, and Assaf Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *CHARME*, pages 129–145, 2005.

[20] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2 :366–381, 2000.

[21] Thomas Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. In *Model Checking Software*, pages 624–624. 2003.

[22] G.J. Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5) :279 –295, may 1997.

[23] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Computer Architecture*, pages 302 –313, apr 1994.

[24] Igor Melatti, Robert Palmer, Geoffrey Sawaya, Yu Yang, Robert M. Kirby, and Ganesh Gopalakrishnan. Parallel and distributed model checking in Eddy. *STTT*, 11(1) :13–25, 2009.

[25] Seungjoon Park and David L. Dill. Protocol verification by aggregation of distributed transactions. In *CAV*, pages 300–310, 1996.

[26] Sylvain Schmitz and Philippe Schnoebolen. Algorithmic Aspects of WQO (Well-Quasy-Ordering) Theory. ESSLLI, 2012.

[27] Murali Talupur and Mark R. Tuttle. Going with the flow : parameterized verification using message flows. In *FMCAD*, pages 10 :1–10 :8, Piscataway, NJ, USA, 2008. IEEE Press.

A. Le protocole German

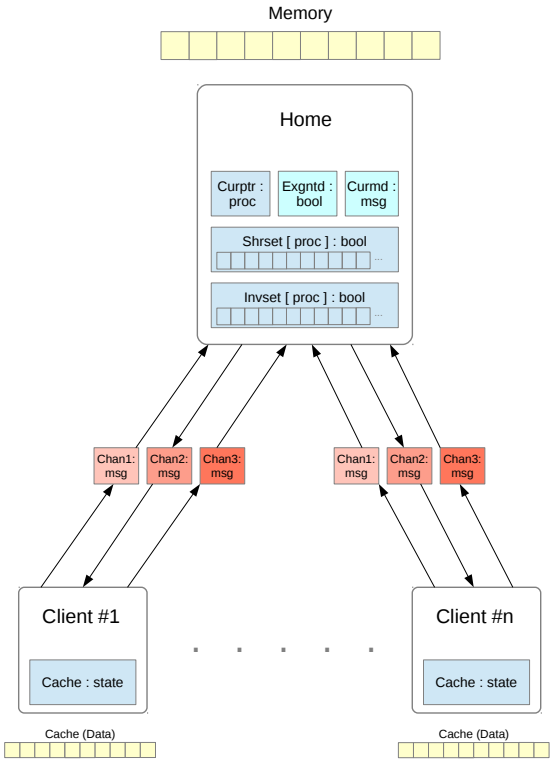

FIGURE 7 – Le protocole German

german.cub

```
type state = Invalid | Shared | Exclusive
type msg = Empty | Reqs | Reqe | Inv | Invack | Gnts | Gnte

var Exgntd : bool
var Curcmd : msg
var CurClient : proc

array Chan1[proc] : msg
array Chan2[proc] : msg
array Chan3[proc] : msg
array Cache[proc] : state
array Invset[proc] : bool
array Shrset[proc] : bool

init (z) {
  Cache[z] = Invalid && Chan1[z] = Empty &&
  Chan2[z] = Empty && Chan3[z] = Empty &&
  Invset[z] = False && Shrset[z] = False &&
  Curcmd=Empty && Exgntd = False }

unsafe (z1 z2)
  { Cache[z1] = Exclusive && Cache[z2] <> Invalid }

transition send_req_shared(n)
requires { Cache[n] = Invalid && Chan1[n] = Empty }
{ Chan1[n] := Reqs; }

transition send_req_exclusive_1(n)
requires { Cache[n] = Invalid && Chan1[n] = Empty }
{ Chan1[n] := Reqe; }

transition send_req_exclusive_2(n)
requires { Cache[n] = Shared && Chan1[n] = Empty }
{ Chan1[n] := Reqe; }

transition recv_req_shared(n)
requires { Curcmd = Empty && Chan1[n] = Reqs }
{ Curcmd := Reqs;
  CurClient := n;
  Invset[j] := case | _ : Shrset[j];
  Chan1[n] := Empty; }

transition recv_req_exclusive(n)
requires { Curcmd = Empty && Chan1[n] = Reqe }
{ Curcmd := Reqe;
  CurClient := n;
  Invset[j] := case | _ : Shrset[j];
  Chan1[n] := Empty; }

transition send_inv_1(n)
requires { Chan2[n] = Empty && Invset[n] = True &&
           Curcmd = Reqe }
{ Chan2[n] := Inv;
  Invset[n] := False; }

transition send_inv_2(n)
requires { Chan2[n] = Empty && Invset[n] = True &&
           Curcmd = Reqs && Exgntd = True}
{ Chan2[n] := Inv;
  Invset[n] := False; }

transition send_invack(n)
requires { Chan2[n] = Inv && Chan3[n] = Empty }
{ Chan2[n] := Empty;
  Chan3[n] := Invack;
  Cache[n] := Invalid; }

transition recv_invack(n)
requires { Chan3[n] = Invack && Curcmd <> Empty }
{ Exgntd := False;
  Chan3[n] := Empty;
  Shrset[n] := False; }

transition send_gnt_shared(n)
requires { CurClient = n && Curcmd = Reqs &&
           Exgntd = False && Chan2[n] = Empty }
{ Curcmd := Empty;
  Chan2[n] := Gnts;
  Shrset[n] := True; }

transition send_gnt_exclusive(n)
requires { CurClient = n && Curcmd = Reqe &&
           Chan2[n] = Empty && Shrset[n] = False &&
           forall_other j. Shrset[j] = False }
{ Curcmd := Empty;
  Exgntd := True ;
  Chan2[n] := Gnte;
  Shrset[n] := True; }

transition recv_gnt_shared(n)
requires { Chan2[n] = Gnts }
{ Cache[n] := Shared;
  Chan2[n] := Empty; }

transition recv_gnt_exclusive(n)
requires { Chan2[n] = Gnte }
{ Cache[n] := Exclusive;
  Chan2[n] := Empty; }
```